15-213 Recitation 11: 11/18/02

Outline

- Robust I/O
- Chapter 11 Practice Problems

What's Left

- L6 *Malloc* Due: Thursday, Nov. 21
- L7 *Proxy* Due: Thursday, Dec. 5
- Final: Tuesday, Dec. 17, 8:30am, Porter Hall 100

Annie Luo

e-mail:

luluo@cs.cmu.edu

Office Hours:

Thursday 6:00 – 7:00

Wean 8402

Why Use Robust I/O

- Handles interrupted system calls
  - e.g. signal handlers
- Handles short counts
  - encountering end-of-file (EOF) on reads (disk files)
  - reading text lines from a terminal
  - reading and writing network sockets or Unix pipes
- Useful in network programs:
  - subject to short counts
  - internal buffering constraints
  - long network delays
  - unreliable

Rio: Unbuffered Input/Output

- Transfer data directly between memory and a file
- No application level buffering
- Useful for reading/writing binary data to/from networks

`ssize_t rio_readn(int fd, void* usrbuf, size_t n)`

- reads **`n`** bytes from **`fd`** and put into **`usrbuf`**
- only returns short count on EOF

`ssize_t rio_writen(int fd, void* usrbuf, size_t n)`

- writes **`n`** bytes from **`usrbuf`** to file **`fd`**
- never returns short count

Rio: Buffered Input

`void rio_readinitb(rio_t* rp, int fd);`

- called only once per open descriptor
- associate **`fd`** with a read buffer **`rp`**

`ssize_t rio_readlineb(rio_t* rp, void* usrbuf, size_t maxlen);`

- for reading lines from text file only
- read a line(stop on **`'\n'`**) or **`maxlen-1`** chars from file **`rp`** to **`usrbuf`**
- terminate the text line with null (zero) character
- returns number of chars read

`ssize_t rio_readnb(rio_t* rp, void* usrbuf, size_t n);`

- for both text and binary files
- reads **`n`** bytes from **`rp`** into **`usrbuf`**
- result string is NOT null-terminated
- returns number of chars read

rio_readlineb

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
{
  int    n, rc;
  char   c, *bufp = usrbuf;

  for (n = 1; n < maxlen; n++) {
    if ((rc = rio_read(rp, &c, 1)) == 1) {
      *bufp++ = c;
      if (c == '\n')
        break;
      }
      else if (rc == 0) {
        if (n == 1)
          return 0;       /* EOF, no data read */
        else
          break;          /* EOF, some data was read */
      }
      else
        return -1;        /* error */
  }

  *bufp = 0;
  return n;
}
```

Interleaving RIO Read Functions

- Do not interleave calls on the same file descriptor between the two sets of functions
- Within each set it is ok

```
rio_readinitb
rio_readlineb
rio_readnb
```

```
rio_readn
rio_writen
```

- Why?

Rio Error Checking

- All functions have upper case equivalents (**`Rio_readn`**, etc.), which call **`unix_error`** if the function encounters an error
  - short reads are not errors
  - wrappers also handle interrupted system calls
  - but they do not ignore EPIPE errors, which are not fatal errors for Lab 7

Problems from Chapter 11

- 11.1
- 11.2
- 11.3
- 11.4
- 11.5

Problem 11.1

- What is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

Answer to Problem 11.1

- Default descriptors:
  - **`stdin`** (descriptor 0)
  - **`stdout`** (descriptor 1)
  - **`stderr`** (descriptor 2)

- **open** always returns *lowest, unopened* descriptor
- First **open** returns 3, **`close`** frees it
- Second **open** also returns 3
- Output of the program:

```
fd2 = 3
```

Kernel Representation of Open Files

- Descriptor table
  - one per process
  - children inherit from parent
- Open file table
  - the set of all open files
  - shared by all processes
  - **`refcnt`**, count of file descriptors pointing to each entry
- V-node table
  - contains information in the **`stat`** structure
  - shared by all processes

Problem 11.2

- Suppose that the disk file **`foobar.txt`** consists of six ASCII characters "**`foobar`**". Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;
    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Answer to Problem 11.2

- Two descriptors **`fd1`** and **`fd2`**
- Two open file table entries and separate file positions for **`foobar.txt`**

- The read from **`fd2`** also reads the first byte of **`foobar.txt`**
- So, the output is

  `c = f`

  and NOT

  `c = o`

Problem 11.3

- As before, suppose the disk file **`foobar.txt`** consists of six ASCII characters "**`foobar`**". Then what is the output of the following program?

```
#include "csapp.h"
int main()
{
    int fd;
    char c;
    fd = Open("foobar.txt", O_RDONLY, 0);
    if(Fork() == 0) {
        Read(fd, &c, 1);
        exit(0);
    }
    Wait(NULL);
    Read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Answer to Problem 11.3

- Child inherits the parent's descriptor table
- Child and parent share an open table entry (`refcnt == 2`)
- They share a file position!

- The output is

```
c = o
```

Problem 11.4

- How would you use **`dup2`** to reflect **`stdin`** to descriptor 5?

- **`int dup2(int oldfd, int newfd);`**
  - Copies descriptor table entry **`oldfd`** to descriptor table entry **`newfd`**

Answer to Problem 11.4

```
dup2(5,0);
```

Or

```
dup2(5,STDIN_FILENO);
```

Problem 11.5

- Assuming that the disk file **`foobar.txt`** consists of six ASCII characters "**`foobar`**". Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;
    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd2, &c, 1);
    Dup2(fd2, fd1);
    Read(fd1, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Answer to Problem 11.5

- We are redirecting **`fd1`** to **`fd2`**. So the second **`Read`** uses the file position offset of **`fd2`**.

`c = o`